Deep Learning Practical Work 1cd

Introduction

In this project we analyzed the fundamental blocks of CNNs and we have implemented an initial network trained on the CIFAR-10 dataset. We also explored several techniques to develop the learning process of the network, adding normalization, data augmentation, different stochastic gradient descent variants, dropout, and batch normalization.

We first analyzed the working mechanism of convolutional layers and pooling layers.
Afterwards we trained a model from scratch on the CIFAR-10 dataset. Our model, inspired by the AlexNet architecture, comprises convolutional and max-pooling layers, fully-connected layers, followed by a softmax function for classification.
In the final stage, our aim is to improve the model's performance. We'll explore several techniques to do this:

- Standardization: normalizing images by calculating mean values and standard deviations (for each channel).
- Data augmentation: generating variations of existing images through random cropping and horizontal symmetry, etc.
- Variants on the optimization algorithm: improved stochastic gradient descent for better convergence (Adam, ...)
- Dropout layers: to prevent overfitting

1. Considering a single convolution filter of padding p, stride s and kernel size k, for an input of size x × y × z what will be the output size ? How much weight is there to learn ? How much weight would it have taken to learn if a fully-connected layer were to produce an output of the same size ?

The output size of a convolutional layer can be calculated using the following formula:

1. **Output Size**:

   For a single convolution filter with padding p, stride s, and kernel size k applied to an input of size x × y × z, the output size will be:

   $x' = [\frac{(x+2p-k)}{s}] + 1$
   $y' = [\frac{(y+2p-k)}{s}] + 1$

   The output depth is determined by the number of filters in the layer. Each filter produces one channel in the output. Notice that, the "2p" is inteded to be as define in the pytorch documentation. See reference: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html .
2. **Number of Weights to Learn**:
   The number of weights to learn in a convolutional layer is determined by the kernel size and the number of input channels. For each filter, you have k x k x c weights, where k is the kernel size, and c is the number of input channels.

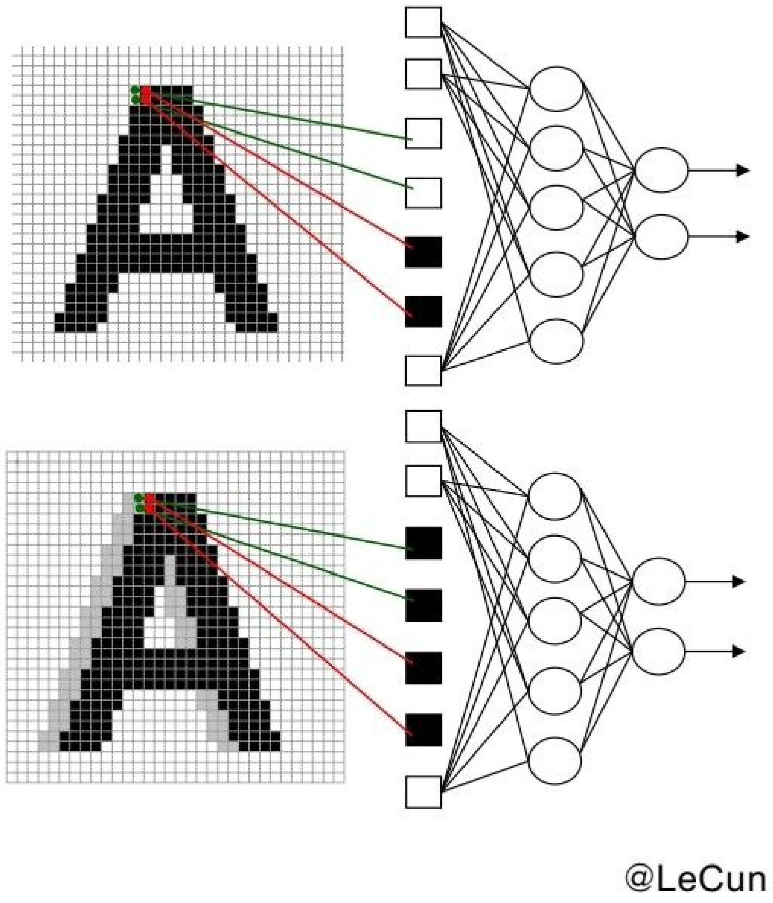
   So, the total number of weights for one filter is k×k×c, and if you have N filters in the layer, you'll have N × k × k × c weights to learn.
3. **Comparison to Fully-Connected Layer**:
   If a fully-connected layer were to produce an output of the same size as the convolutional layer, you would have to connect each input neuron to each output neuron. In this case, the number of weights in the fully-connected layer would be x × y × z for each output neuron.

2. What are the advantages of convolution over fully-connected layers ? What is its main limit ?

There are two main problems with the brute-force fully connected approach for images:

1. In the case of the **MLP** with only fully connected layers, an ideal representation should maintain similarity for small variations/deformations in the input. Conversely, for larger variations, the representation should be notably different from the original input. This feature is crucial especially when extracting features from the input, as is the case in semantic segmentation. However, in fully connected neural networks, this is not the case because small changes in the input can lead to significant alterations in the network's output. For instance, if we translate an image slightly to the left or right we might cause a fully connected NN to fail in recognizing the same pattern as in the original position.

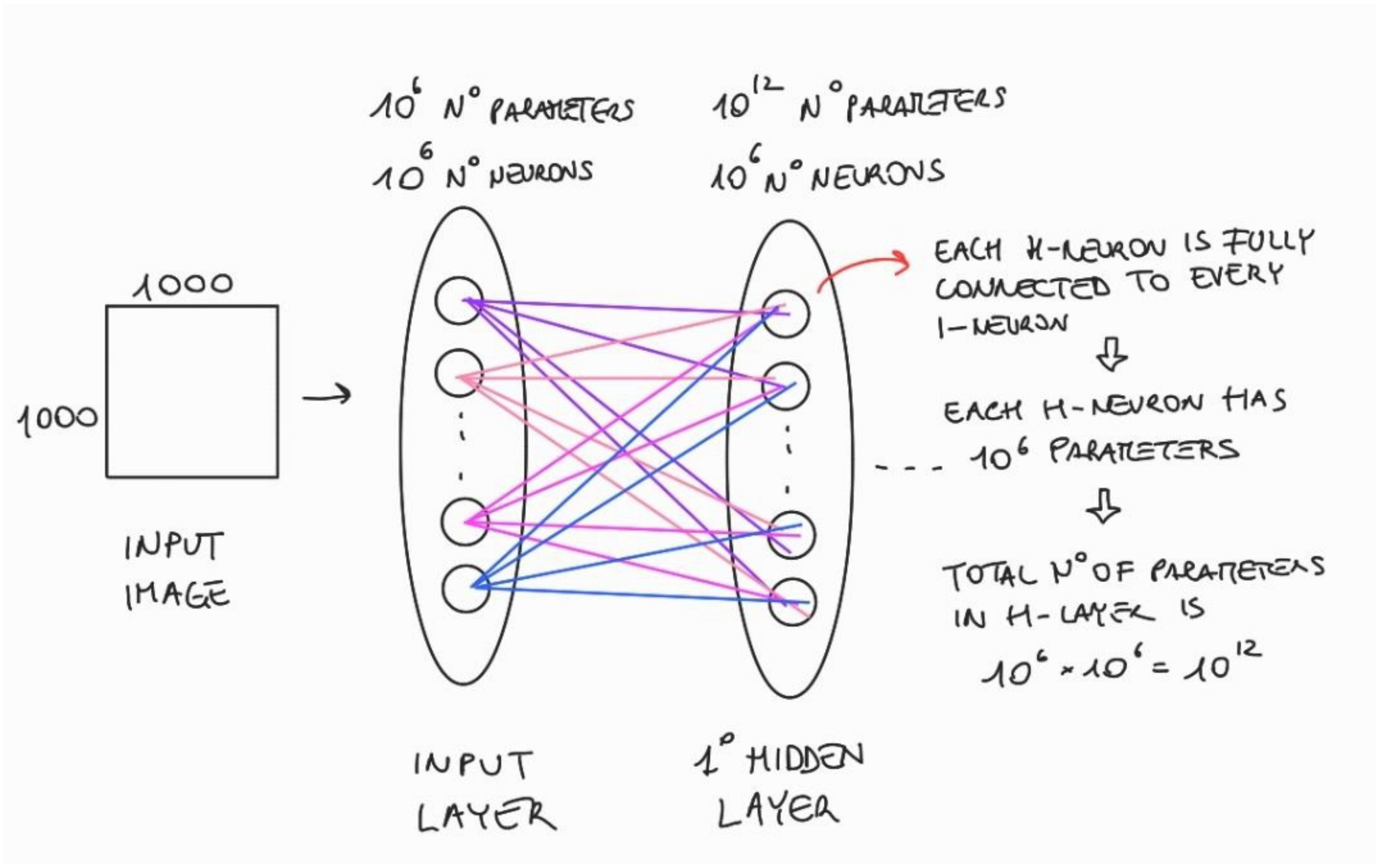
   ```
   Consider the example (shown in the figure below): where the green and red dots are near each other,
   and if the pixelated image is flattened, it produces the displayed black and white input vector.
   Shifting the black pixels by two positions to the right results in a significant number of input
   neurons switching from white to black and vice versa (approximately 150 neurons), leading to a
   vastly different image representation.
   ```

   What's even worse is that if you take an input image and randomly permute its pixels, then train it in a fully connected neural network, you would obtain similar results as if you trained it with the non-permuted original image! This implies that the image's structure or arrangement is entirely disregarded.
2. There are a larger number of weights compared to the convolutional layer, making it computationally expensive and susceptible to overfitting, especially when the input size is large.

Example:
If you have an image that is 1000x1000, then after flattening it, you would get an input vector of 10^6 (11 million) parameters. If you then opt to have a hidden layer of the same size as the previous layer, you have a hidden layer comprising 10^6 parameters, and each neuron is fully connected to every preceding neuron, resulting in 10^6 x 10^6 parameters in total for that hidden layer. This quantity of parameters is extensive to store in conventional computers and represents just one layer, posing a scalability

issue.

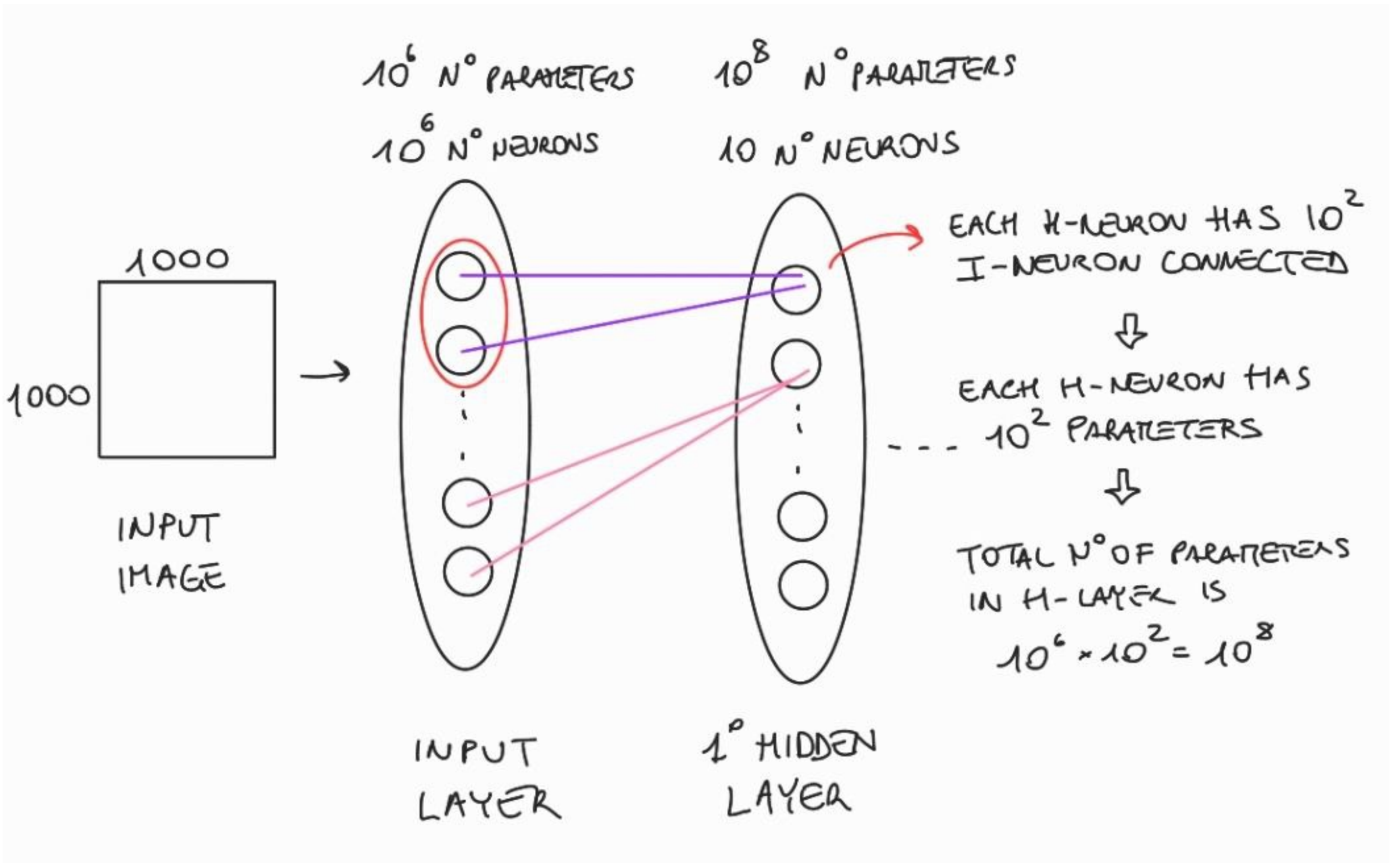


On the contrary, there are two main concepts for the **convolutional layers**:

- The first idea involves creating local connections instead of global connections: each hidden layer of the neural network is not connected to all the neurons in the next layer but only to some of them. This is referred to as sparse connectivity (versus full connectivity), and the weights associated with a local patch of neurons are termed filters or kernels. Another way to express this is that convolutional layers are responsive to a small sub-region of the input space, referred to as a receptive field (analogous to biology).

Example:
If you have an image that is 1000x1000, flattening it results in an input vector of 10^6 (11 million) parameters. If you then choose to have a hidden layer of the same size as the previous layer, resulting in a hidden layer of 10^6 parameters. If this hidden layer is a convolutional layer, we decide to connect a small number of neurons in the input layer to each neuron of the hidden layer. Given that the input is an image, we arbitrarily opt to connect small patches of the image that are 10x10 in size. This implies that each hidden neuron will be linked to 10^2 input neurons. Hence, the total number of parameters for the first hidden layer is 10^6

x 10^2 = 10^8, which is significantly fewer than the results obtained previously with the fully connected neural network.


10^6 N° PARAMETERS
10^6 N° NEURONS
10^8 N° PARAMETERS
10 N° NEURONS
1000
1000
INPUT IMAGE
EACH H-NEURON HAS 10^2 I-NEURON CONNECTED
EACH H-NEURON HAS 10^2 PARAMETERS
TOTAL N° OF PARAMETERS IN H-LAYER IS 10^6 × 10^2 = 10^8
INPUT LAYER
1° HIDDEN LAYER

- Secondly, there's the concept of shared weights, signifying the utilization of the same set of weights (also termed a kernel or filter) across multiple spatial locations in an input feature map. Instead of having distinct weight sets for each location in the input, a single set of weights is employed across the entire input. This is vital to achieve stability, which is unattainable in fully connected layers, providing equivariance to translation. When you shift the original image and apply convolution, you obtain the same results as the original image but with the applied shifting. This implies that objects or patterns might appear at different positions in an image, yet their identity remains consistent. Through weight sharing, Convolutional Neural Networks (CNNs) can recognize the same feature irrespective of its position in the input, preserving strong spatial information. Conversely, in contrast to what was mentioned for the fully connected layers, if you permute the input, the output of the convolutional layer will significantly differ from the output generated by the non-permuted input.

When training your model, your objective is to learn the kernels. Therefore, in CNNs, the weights represent the kernels.

Applying one convolution with a specific kernel to an image (or feature map) results in one image (or feature map) as the output. Typically, multiple kernels are applied to output several feature maps, providing a more comprehensive description and increasing the number of parameters. If n convolutions are applied between an input layer and a subsequent hidden layer, n outputs are obtained (refer to image slides 18-19). Using multiple filters doesn't significantly increase the number of weights (due to weight sharing constraints), but it does amplify the number of neurons, potentially leading to memory issues.

When applying convolution to a given input and aiming for the output to have the same dimension, configuring the padding and stride is essential.

In some tasks, especially in classification, it's desirable for the convolutional layer not only to be equivariant to translation operations but also invariant. The primary distinction between invariance and equivariance lies in the fact that equivariance allows the network to generalize edge, texture, and shape detection in different locations, while invariance allows the precise location of the detected features to matter less. To achieve more local invariance, pooling operations can be added to the network. Refer to question 3 for more details.

3. Why do we use spatial pooling ?

In certain tasks, especially in classification, one desires the convolutional layer not only to be equivariant to translation operations but also invariant. The primary distinction between invariance and equivariance is that equivariance allows the network to generalize edge, texture, and shape detection in different locations, while invariance enables the precise location of detected features to matter less. To achieve more local invariance in the results, pooling operations can be added to the network. Pooling aggregates the values in the input feature map to summarize information within local regions. There are several ways to pool the values (max, average, etc.). After translating your input image, applying convolution results in a shifted output (as convolutional layers are equivariant). To correct these shifts, a pooling operation can be employed to yield the same outcome as if the same pipeline were computed with a non-shifted input image.

Another significant use of spatial pooling is to consistently reduce the size of the feature map across the network layers, generating a representation of salient features.

When delving into the details of pooling operations, pooling is applied to each channel of the input. Various parameters of the pooling operation can be adjusted, such as stride or padding. Typically, after pooling, the objective is not to attain an output image of the same size as the input image, so a stride greater than 1 is chosen, downsizing the image.

4. Suppose we try to compute the output of a classical convolutional network (for example the one in Figure 2) for an input image larger than the initially planned size (224 × 224 in the example). Can we (without modifying the image) use all or part of the layers of the network on this image ?

No, you cannot use **all** parts of the layers. Taking the VGG16 network as an example, if you provide it with an input image larger than 224x224 (for instance, a 400x400 image), every convolutional layer and max pooling operation will yield larger outputs (for example, the output of the first convolutional layer would be 400x400x64). However, these outputs won't fit into the fully connected layer. This limitation occurs because when designing your model, you need to specify and fix the size of the input and output for the fully connected layer.

```
nn.Linear(input_size, output_size)
```

Instead, for convolutional layer, you don't specify the input or output size, but just the number of channels (= number of filters) in input and output and the size of the kernel:

```
nn.Conv2d(in_channels, out_channels, kernel_size)
```

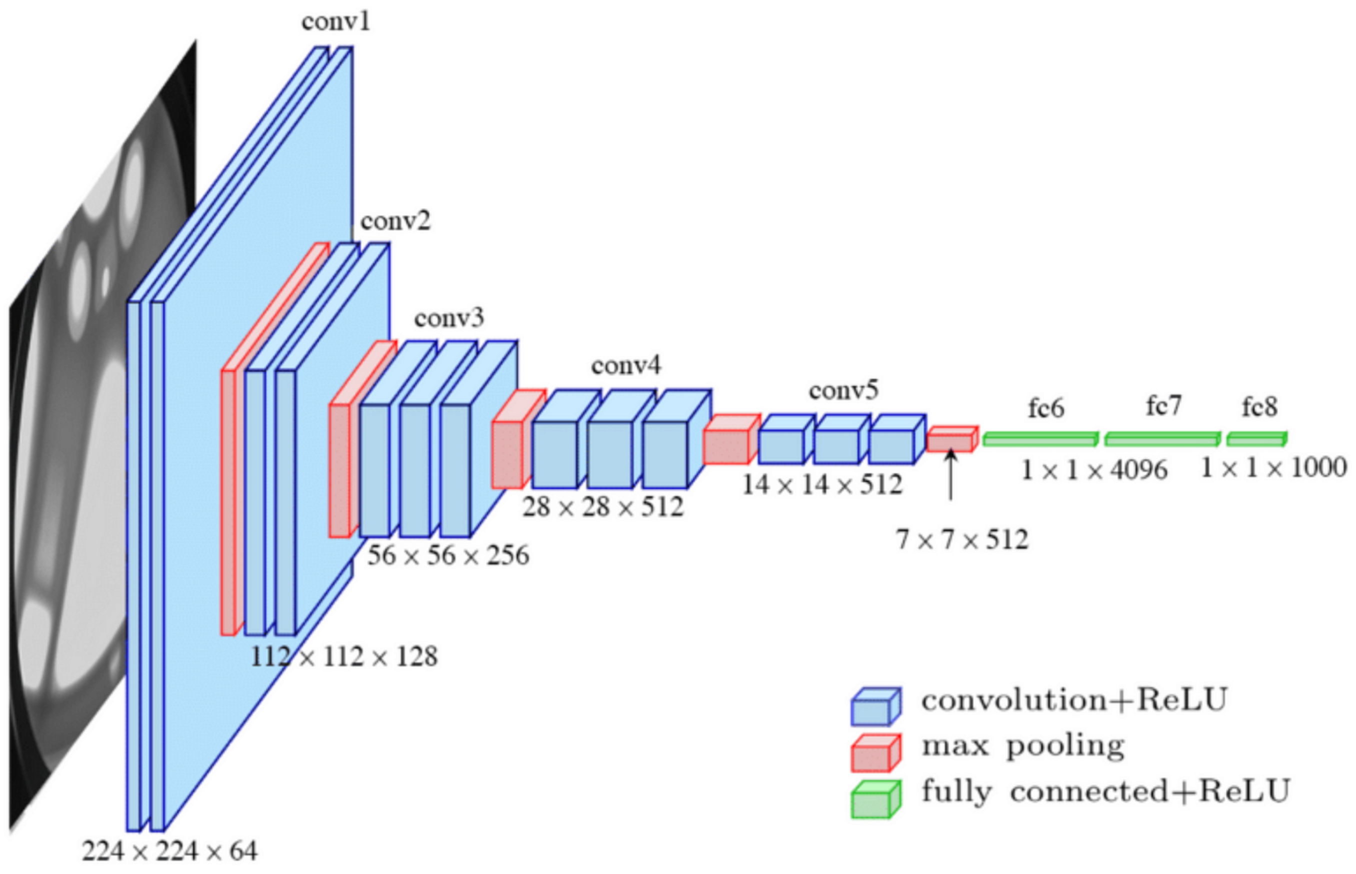
This is also the case for Max Pooling operations.

5. Show that we can analyze fully-connected layers as particular convolutions.

Mathematically, the operation of a fully-connected layer and a convolutional layer with a kernel size equal to the input size is the same. In both cases, each output unit computes a weighted sum of all the input values.

- In a fully-connected layer, if we consider the weight matrix W as the convolutional kernel, and the input as a 2D or 3D grid, then the output is given by $O=W \cdot I$, where O is the output, W is the weight matrix, and I is the input.
- In a convolutional layer with a kernel size matching the input size, the convolution operation is represented as $O=I \ast K$, where O is the output, I is the input, and K is the convolutional kernel. If the size of K is the same as the size of W, zero padding and an arbitrary stride, this is equivalent to the fully-connected layer operation.
  Another method would be do a convolution with 1x1 kernels.

6. Suppose that we therefore replace fully-connected by their equivalent in convolutions, answer again the question 4. If we can calculate the output, what is its shape and interest ?



We aim to replace the FC layers with their equivalent convolutional layers.

The FC6 layer with (c=4096) observing an input volume of size (7×7×512) can be expressed equivalently as a CONV layer with (k=7), (P=0), (S=1), (c=4096), where (k) represents the kernel size, (P) denotes the padding, (S) signifies the stride, and (c) indicates the number of filters. Essentially, we're setting the filter size to match the exact size of the input volume, resulting in an output of (1×1×4096) since only a single depth column "fits" across the input volume, yielding the same outcome as the original FC layer.

Subsequently, the fc7 layer, also with (c=4096), will possess an input volume size of (1×1×4096). Its equivalent convolutional layer would have (k=1), (P=0), (S=1), (c=4096).

Following this, the fc7 layer with (c=1000) (representing the number of classes in ImageNet) will feature an input volume size of (1×1×4096). The equivalent convolutional layer would have (k=1), (P=0), (S=1), (c=1000).

When all the layers are convolutional, the neural network can be applied to images of any size. The output size of the network is dependent on the size of the input images used. However, if the network is applied to smaller images, the output will be empty because the images are insufficiently large to apply the learned filter. Conversely, for larger images, the network's output will not be a single number but a 2D array.

source

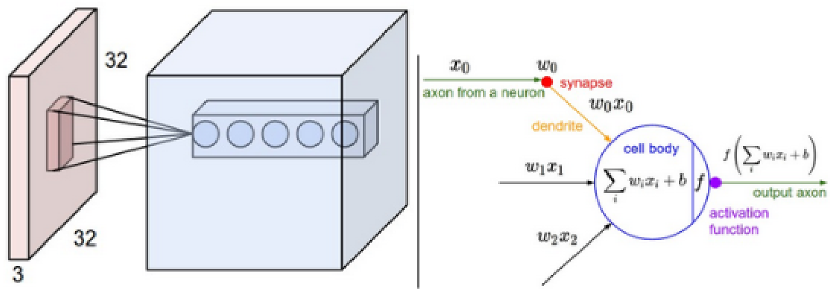
7. We call the receptive field of a neuron the set of pixels of the image on which the output of this neuron depends. What are the sizes of the receptive fields of the neurons of the first and second convolutional layers ? Can you imagine what happens to the deeper layers ? How to interpret it ?

- As you move deeper into the network, the receptive field size continues to increase. This is primarily due to the stacking of convolutional layers and pooling layers with larger windows or strides.
- Deeper layers have progressively larger receptive fields, allowing them to capture more global information and high-level features in the input image.

- This increase in receptive field size is a fundamental aspect of hierarchical feature learning in deep CNNs. Features learned in deeper layers tend to represent more complex and abstract patterns.

On the output of the first convolutional layer, the size of the pixel's receptive field is equal to the size of the convolution filter. By applying a second convolution immediately, the size of the pixel's receptive field becomes $(k2-1)s1+k1$.

The size of the receptive field increases with depth, which means that the early layers have low-level features, and the later layers have higher-level features.



reference

8. For convolutions, we want to keep the same spatial dimensions at the output as at the input. What padding and stride values are needed ?

Considering the equation of the first answer:
$x' = [\frac{(x+2p-k)}{s}]+1$
if we assume the condition $x = x'$:

$$x = [\frac{(x+2p-k)}{s}]+1$$

to obtain $x = x$, we need to have $s = 1$, because it is in the denominator.
Therefore:

$$x = x+2p-k+1$$
$$0 = 2p-k+1$$
$$p = \frac{k-1}{2}$$

If we assume that k=5, we obtain the solution $p = 2, s = 1$

Notice that, to maintain the size of the images when applying a convolution, a stride of 1 is used, along with padding based on the size of the kernel 'k':

- For an odd 'k', a padding of size $\frac{k-1}{2}$ is appropriate.
- For an even 'k,' you should also use a padding of size $\frac{k-1}{2}$, but this results in a non-integer padding size. That's why, in practice, odd-sized kernels are used.

9. For max poolings, we want to reduce the spatial dimensions by a factor of 2. What padding and stride values are needed ?

Considering the equation of the relation between the input size and the output size after a maxpooling operation:
$x' = [\frac{(x+2p-k)}{s}]+1$
if we assume the condition $x' = \frac{x}{2}$:

$$\frac{x}{2} = [\frac{(x+2p-k)}{s}]+1$$

$$x = 2[\frac{(x+2p-k)}{s}]+2$$

to remove the multiplied factor of 2 we need to have $s = 2$.
Therefore:

$$x = x + 2p - k + 2$$
$$0 = 2p - k + 2$$
$$p = \frac{k-2}{2}$$

If we assume that k=2, we obtain the solution $p = 0, s = 2$

10. For each layer, indicate the output size and the number of weights to learn. Comment on this repartition.

You can see it in the code, or otherwise here:

Layer	Size of the output	Number of weights
Input	3x32x32	0
conv1	32x32x32	5x5x3x32 + 32 = 2432
pool1	32x16x16	0
conv2	64x16x16	5x5x32x64 +64 = 51264
pool2	64x8x8	0
conv3	64x8x8	5x5x64x64 + 64= 102464
pool3	64x4x4	0
fc4	1000	1000x64x4x4 +1000 = 1025000
fc5	10	10x1000 +10 = 10010

Total number of parameters = 1 191 170

11. What is the total number of weights to learn ? Compare that to the number of examples.

To calculate the total number of weights to learn in a neural network, you need to sum up the weights in all the layers of the network.

1. Convolutional Layers:
   - For each convolutional layer, you need to consider the number of input channels, the number of output channels (filters), the size of each filter, and whether or not there is a bias term. The formula for calculating the number of weights in a convolutional layer is:
     Number of Weights = (Input Channels *Output Channels* Filter Height * Filter Width) + (Output Channels if there's a bias term)
2. Fully Connected Layers:
   - For each fully connected (linear) layer, you need to consider the number of input neurons and the number of output neurons, including bias terms. The formula for calculating the number of weights in a fully connected layer is:
     Number of Weights = (Input Neurons * Output Neurons) + (Output Neurons if there's a bias term)
3. Sum up the number of weights for all layers to find the total number of weights in your network.

Method by hand (you can see also look at the previous table):

1. Convolutional Layer 1:
   - Input Channels: `3` (assuming color images)
   - Output Channels (Filters): `32`
   - Filter Height: `5`
   - Filter Width: `5`

    - Bias Term: Yes
    - Number of Weights = `(3 * 32 * 5 * 5) + 32 = 2432`
2. Convolutional Layer 2:
    - Input Channels: `32` (output from the previous layer)
    - Output Channels (Filters): `64`
    - Filter Height: `5`
    - Filter Width: `5`
    - Bias Term: Yes
    - Number of Weights = `(32 * 64 * 5 * 5) + 64 = 51264`
3. Convolutional Layer 3:
    - Input Channels: `64` (output from the previous layer)
    - Output Channels (Filters): `64`
    - Filter Height: `5`
    - Filter Width: `5`
    - Bias Term: Yes
    - Number of Weights = `(64 * 64 * 5 * 5) + 64 = 102464`
4. Fully Connected Layer 4:
    - Input Neurons: `64 * 4 * 4` (output from the last convolutional layer, which is 64 channels, each of size 4x4)
    - Output Neurons: `1000`
    - Bias Term: Yes
    - Number of Weights = `(64 * 4 * 4 * 1000) + 1000 = 1025000`
5. Fully Connected Layer 5:
    - Input Neurons: `1000` (output from the previous fully connected layer)
    - Output Neurons: `10` (assuming this is a classification layer)
    - Bias Term: Yes
    - Number of Weights = `(1000 * 10) + 10 = 10010`

Now, let's sum up the number of weights from each layer:

- Convolutional Layers: `2432 + 51264 + 102464 = 156160` weights
- Fully Connected Layers: `1025000 + 10010 = 1035010` weights

Total Number of Weights in the `ConvNet2` model: `156160 + 1035010 = 1191170` weights.

Method coded:
look at count_parameters() function in project1cd2.py:

```
def count_parameters(self):
        total_params = 0
        for param in self.parameters():
                total_params += param.numel()
        return total_params
```

Comparing the number of the weights to be learned (1191170) with the number of the images in the training dataset (50000), we think that the number of images is too small to learn such number of weights and this could lead to underfitting or overfitting.

12. Compare the number of parameters to learn with that of the BoW and SVM approach.

In the previously employed Bag of Words (BoW) approach coupled with Support Vector Machine (SVM) classification, utilizing a dictionary comprising 1000 Scale-Invariant Feature Transform (SIFT) descriptors, the total number of parameters to be learned was approximately 128,000. This method involved a series of hyperparameters that needed to be carefully configured, including the size of the learned dictionary, the tuning of the 'C' constant in the SVM classifier, and the specific type of SIFT descriptors used.

In contrast, when employing a convolutional neural network (CNN), we encounter a substantially higher number of parameters to train. To be precise, the CNN demands learning tenfold more weights compared to the BoW+SVM approach. Additionally, the neural architecture introduces a set of hyperparameters to consider, but it offers a unique advantage. The CNN operates on an end-to-end learning paradigm, which enables it to automatically acquire and adapt the feature extraction process.

This means that, with a CNN, the model learns not only how to classify images but also how to extract and represent the most relevant features from the data, whereas the BoW+SVM approach relies on predefined features and requires fine-tuning of its parameters to achieve optimal performance. The increase in parameter count in the CNN is a trade-off for this added capacity to learn a broader range of features directly from the data.

13. Read and test the code provided. You can start the training with this command : main (batch_size, lr, epochs, cuda = True)

In the code

14. In the provided code, what is the major difference between the way to calculate loss and accuracy in train and in test (other than the the difference in data) ?

They both use AverageMeter().

```
#IN EPOCH()
# indicates whether the model is in eval or train mode (some layers behave differently in train and
eval)
model.eval() if optimizer is None else model.train()
# compute metrics
prec1, prec5 = accuracy(output, target, topk=(1, 5))
batch_time = time.time() - tic
tic = time.time()

# update
avg_loss.update(loss.item())
avg_top1_acc.update(prec1.item())
avg_top5_acc.update(prec5.item())
avg_batch_time.update(batch_time)
if optimizer:
        loss_plot.update(avg_loss.val)


# IN MAIN
# Train phase
top1_acc, avg_top5_acc, loss = epoch(train, model, criterion, optimizer, cuda)
# Test phase
top1_acc_test, top5_acc_test, loss_test = epoch(test, model, criterion, cuda=cuda)
```

In the epoch() function, if the parameter "optimizer" is None, then it calls model.eval():

```
def eval(self):
        r"""Sets the module in evaluation mode."""
        return self.train(False)
```

Otherwise, it calls method model.train():

```
def train(self, mode=True):
        r"""Sets the model in training mode."""
        self.training = mode
        for module in self.children():
                module.train(mode)
        return self
```

As demonstrated in the code above, the `model.train()` function sets the modules in the network to the training mode. It informs our model that we are presently in the training phase, allowing certain layers such as dropout and batch normalization, which behave differently based on the current phase, to be activated. Conversely, `model.eval()` performs the opposite function. During the training mode, BatchNorm updates a moving average with each new batch, whereas, during the evaluation mode, these updates are suspended. Therefore, upon calling `model.eval()`, these layers are deactivated, ensuring that the model produces its inference as expected.

Moreover, the error displayed during training is an average computed over various batches of data. This implies that it does not represent the final error at the end of an epoch, which is the case during testing.

15. Modify the code to use the CIFAR-10 dataset and implement the architecture requested above. (the class is datasets.CIFAR10). Be careful to make enough epochs so that the model has finished converging.

In the code

16. What are the effects of the learning rate and of the batch-size ?

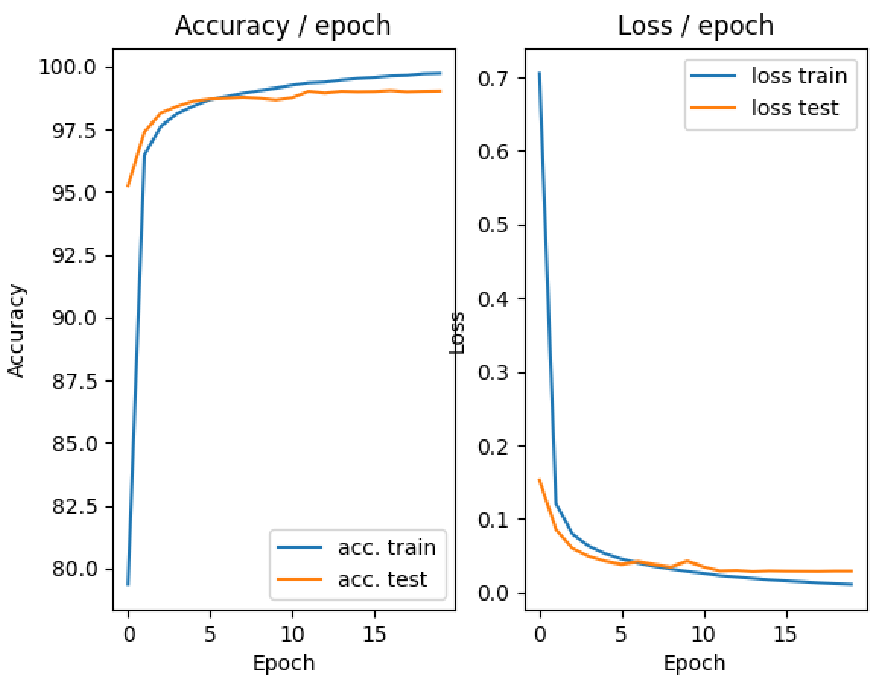
Convergence depends on the choice of η (eta, the learning rate): if it's too small, the model will take a long time to train, and if it's too large, the model may fail to converge. In practice, it's common to decrease η as training progresses.

Moreover, depending on the amount of data used to compute the gradient, we have:

- Batch gradient descent: The gradient is computed over the entire dataset for a single precise update, which can be slow.
- Stochastic gradient descent: It computes the gradient one example at a time with an update after each, making the training faster and suitable for online learning. However, the model may struggle to converge to the global minimum if the gradient step size doesn't decrease over iterations.
- Mini-batch gradient descent: The gradient is computed over a batch of examples, reducing the parameter update variance, resulting in more stable convergence.

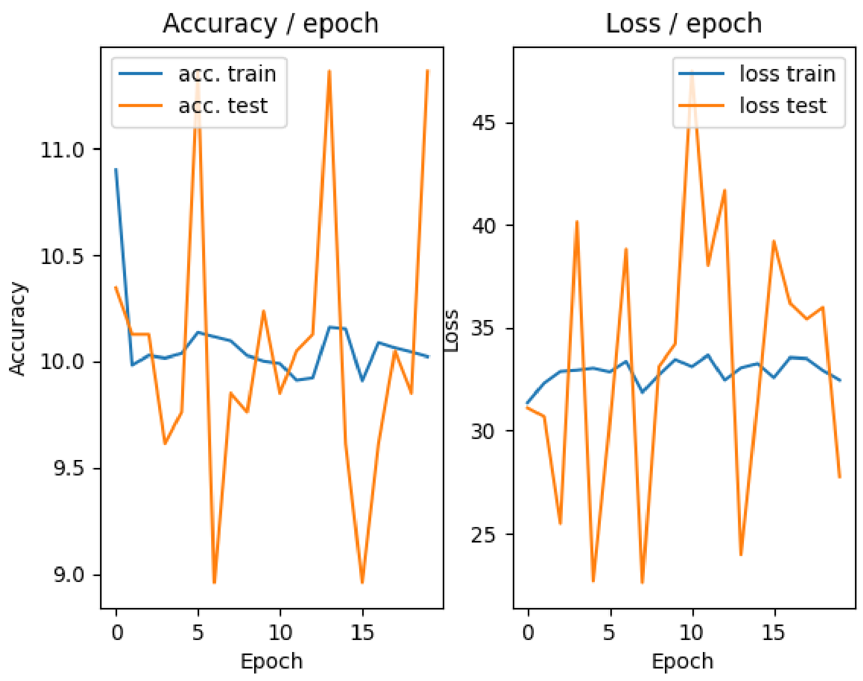
Here we present extended experiments:

Baseline: batch_size=128, lr=0.1, epochs=20, cuda=True

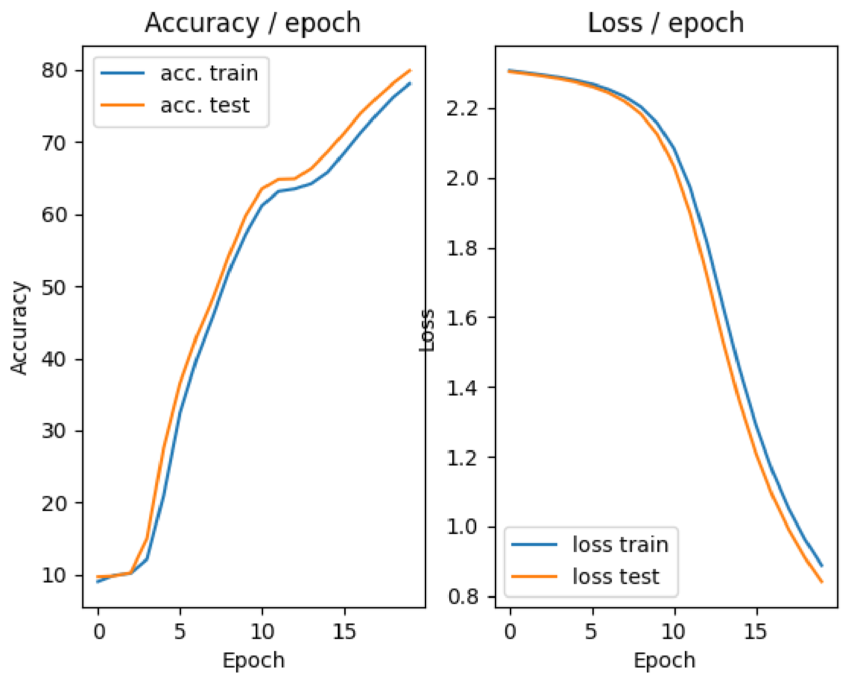


1. Changing Learning Rate:

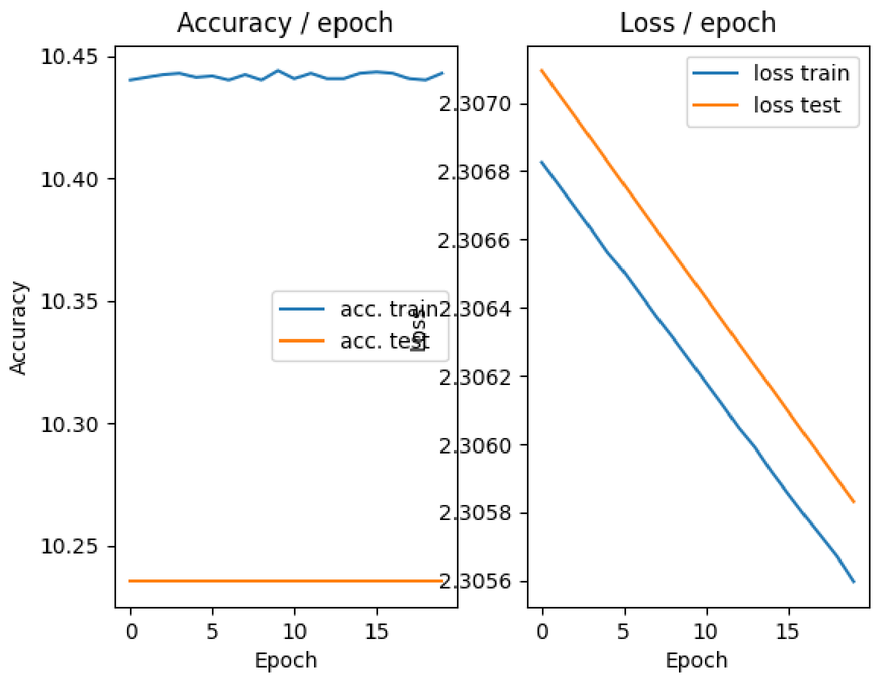
- **Too High:** A high learning rate can lead to rapid convergence, but it may overshoot the optimal weights and lead to divergence. This can result in the loss function increasing rather than decreasing. Training may become unstable, and the model may not converge to a good solution.
  Experiment: epochs=20; lr=1, batch_size=128



- **Too Low:** A low learning rate can make the training process very slow. The model may get stuck in local minima or plateaus, and it may require a large number of epochs to converge. It's also sensitive to noise in the data.
  Experiment: epochs=20; lr=0.001, batch_size=128

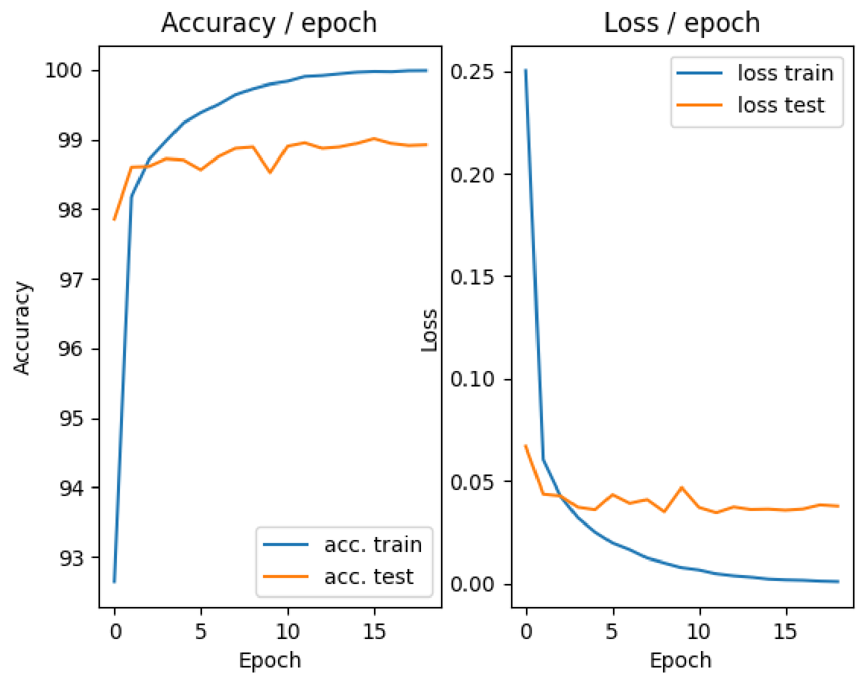


Experiment: epochs=20; lr=0.00001, batch_size=128

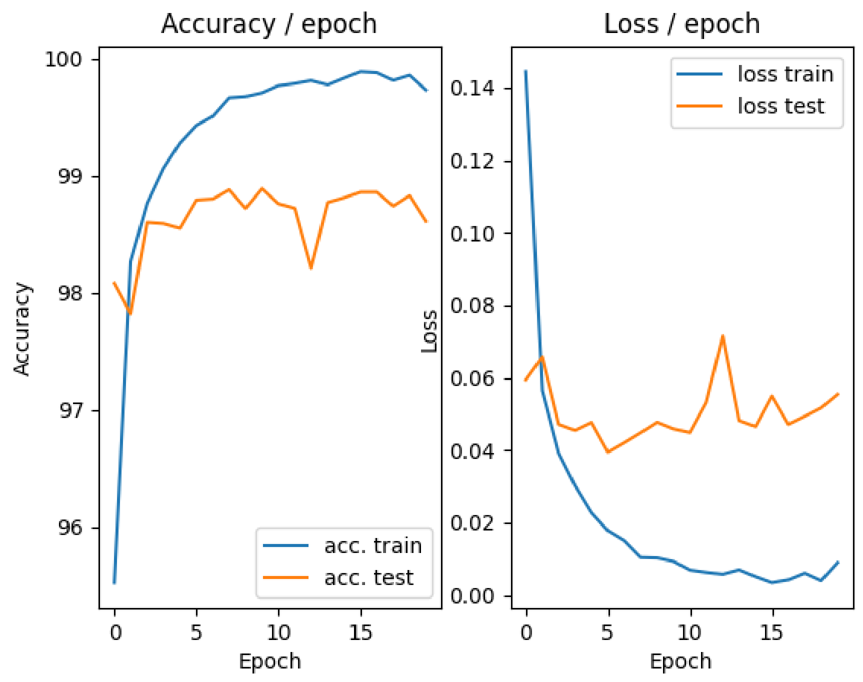


2. Changing Batch Size:

- **Small Batch Size:** Using a small batch size (e.g., 1, 32) can result in noisy updates. It can lead to faster convergence within an epoch, but the model's training can be less stable due to the randomness of each batch. It may also require more epochs to converge, and it may get stuck in local minima.
  Experiment: epochs=20; lr=0.1, batch_size=32

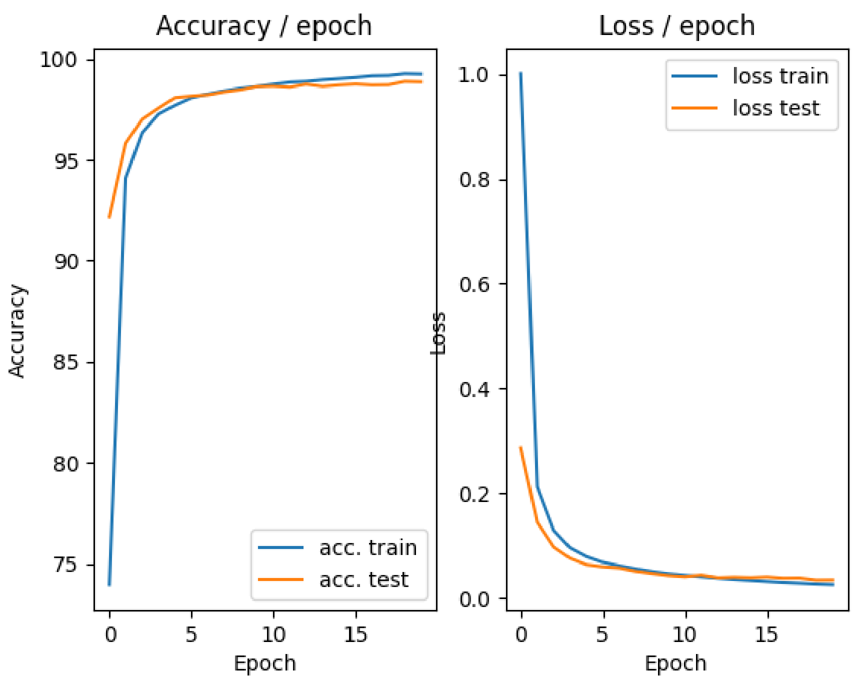


Experiment: epochs=20; lr=0.1, batch_size=8
Time computation: 37seconds per each epoch (which is a lot more than the previous ones)



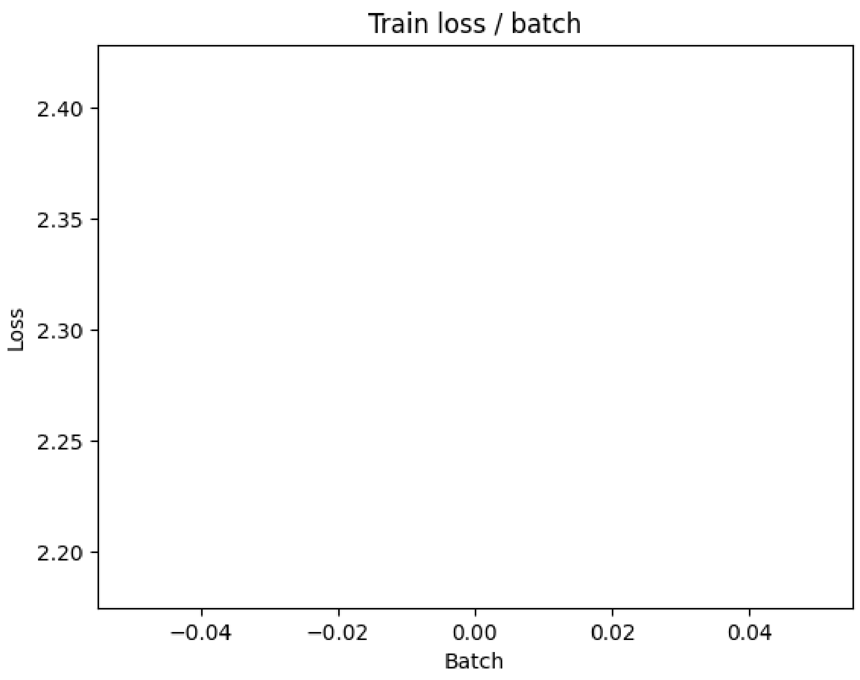
- **Large Batch Size:** A large batch size (e.g., 64, 128, 256) can provide more stable and accurate gradient estimates, resulting in smoother training curves. However, it may require more memory and can be computationally expensive. It might also converge to a wider minima rather than the optimal minima, which could affect generalization.

Experiment: epochs=20; lr=0.1, batch_size=256



17. What is the error at the start of the first epoch, in train and test ? How can you interpret this ?

In the first epoch we get:
[TRAIN Batch 000/469] Time 0.104s (0.104s) Loss 2.3017 (2.3017) Prec@1 7.8 (7.8) Prec@5 50.8 (50.8)



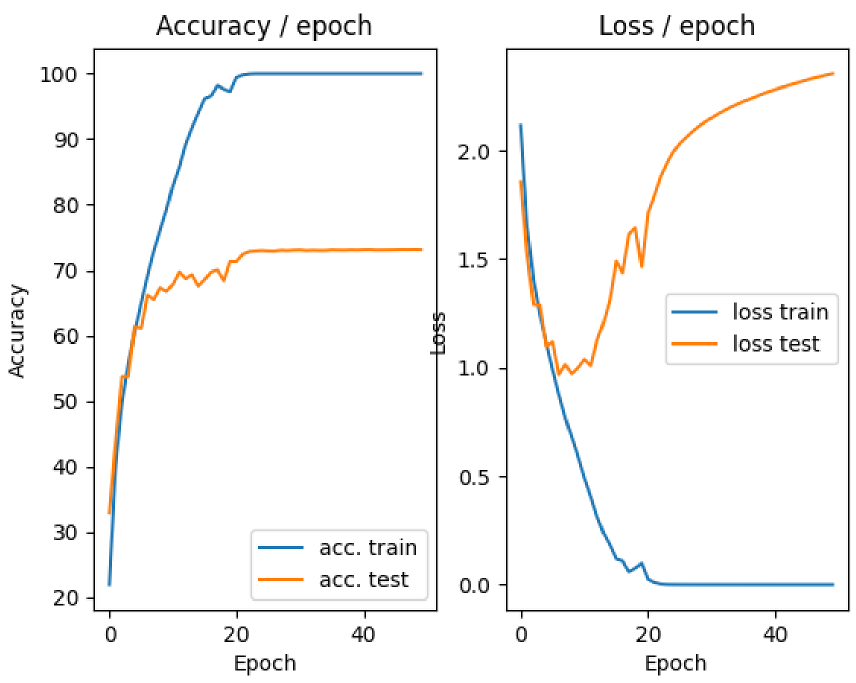
This is because if you create a line plot with just a single value using `plt.plot([single_value])`, it should not display a point or a line. Instead, it should create an empty Cartesian graph without any visible data.
If you would like to see a single point plotted, you would have to modify the parameters of the plot function, for instance, like this:

```
# Creating a point marker for the single value
plt.plot([0], [single_value], marker='o', markersize=5)
```

From the text output, we can see that there is still a loss and accuracy that is calculated, but that is because of the random initialization of the network's weights, since the model is not trained yet. This serves as a kind of baseline: during training, the model should reduce this error. If it doesn't, it indicates that it is struggling to learn.

18. Interpret the results. What’s wrong ? What is this phenomenon ?

Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



These plots display the accuracy percentage over the number of epochs (on the left) and the loss number over the number of epochs (on the right). Our objective is to minimize the loss as much as possible and maximize the accuracy as much as possible. The different metrics are calculated on the train set (in blue) and on the test set (in orange). It's worth noting that the "test set" is not intended to be the set of images from the dataset used to make predictions after the training. Instead, it is used to evaluate our training and is calculated during the training itself. More precisely, we refer to it as the "val test".

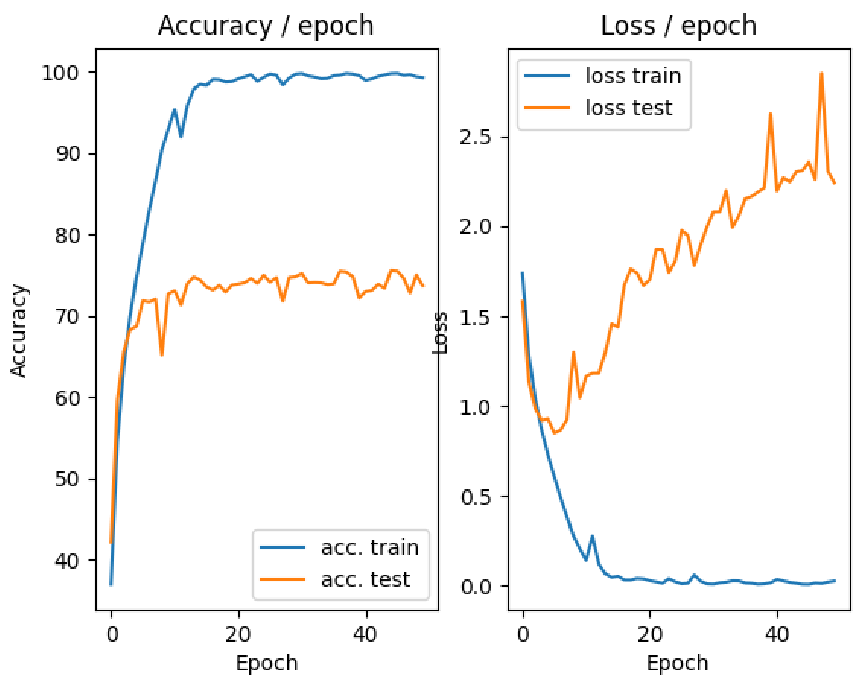
As observed, the accuracy of the train set is increasing, while the loss of the train set is decreasing, which is favorable for us. However, the loss of the test set forms a "U" shape, indicating that at a certain point, it stops decreasing and starts to increase. This phenomenon is known as overfitting, where the model becomes overly specialized to the training data and performs poorly on unseen data. Furthermore, the test accuracy tends to plateau at 70% instead of striving to achieve a better result. Therefore, we should consider modifying our hyperparameters or the architecture of the model, improving the dataset, or using other metrics to address this issue.

This underscores the importance of the evaluation set (referred to here as the test set): if we were to evaluate our model only on the train set, we might mistakenly believe that the model performed well, which is not the case.

3.1 Standardization of examples

19. Describe your experimental results.

Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



The training has shown some improvement compared to before. The test accuracy is slightly higher, exceeding 70%, and the minimum test loss is lower than the previous 1.0. However, we still observe an overfitting phenomenon, and the curves for the test and train sets exhibit noticeable differences.

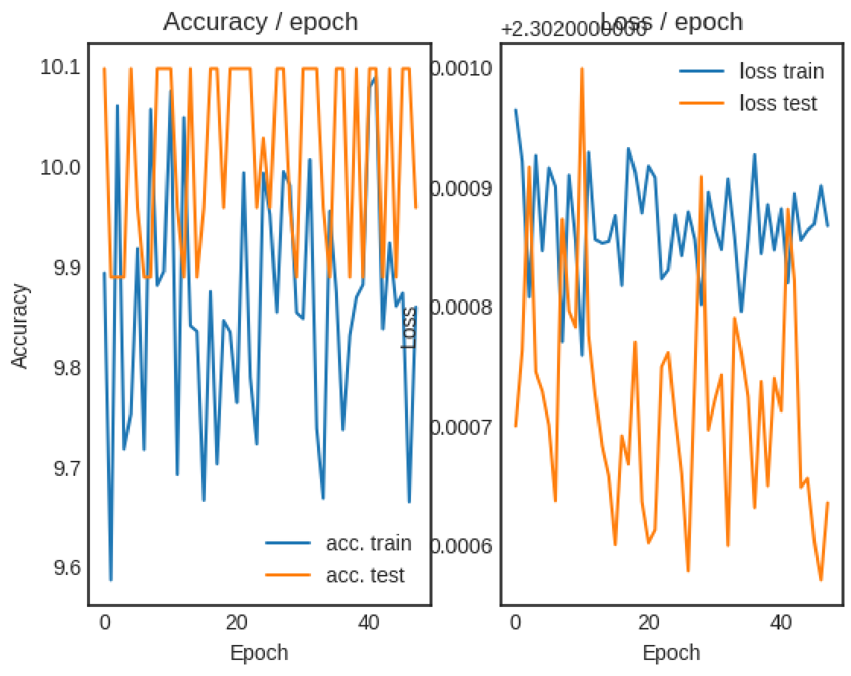
20. Why only calculate the average image on the training examples and normalize the validation examples with the same image ?

Preprocessing should not be learned on the test dataset, as it would bias the obtained performance results.

21. Bonus : There are other normalization schemes that can be more efficient like ZCA normalization. Try other methods, explain the differences and compare them to the one requested.

We tried to perform ZCA normalization: to calculate the ZCA transformation matrix we use PCA by scikit-learn and saved it in an external file. After computing that, we trained the the neural network using the ZCA normalization instead of the standard normalization.
The results that we obtained are very bad compared to the previous ones: the accuracy is very low and oscillates too much.

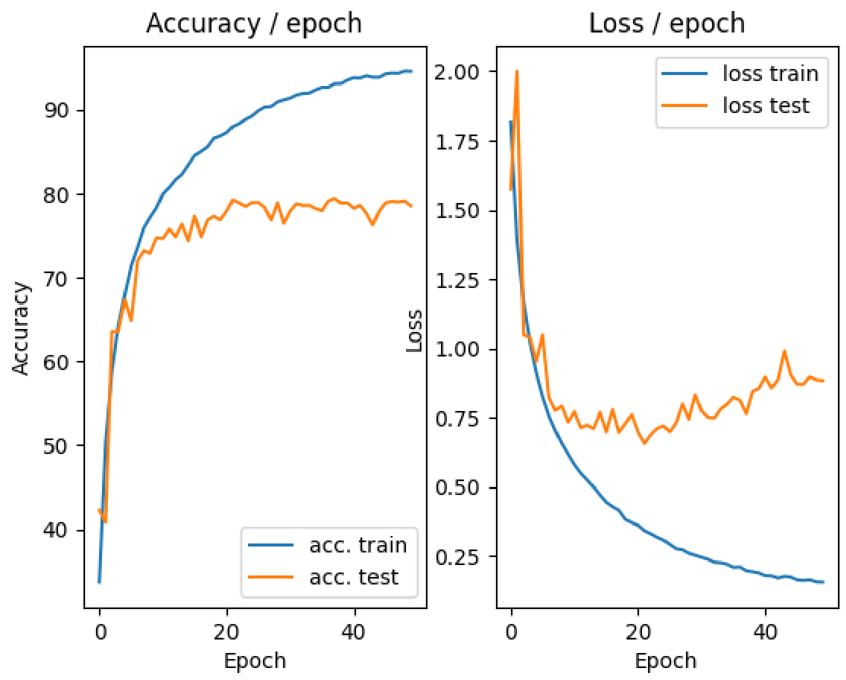
Here we show the results of an experiment with the following hyperparameters batch_size=128, lr=0.1, epochs=50, cuda=True



3.2 Increase in the number of training examples by data increase

22. Describe your experimental results and compare them to previous results.

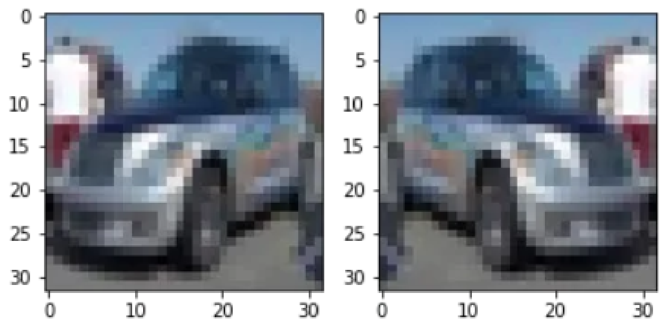
Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



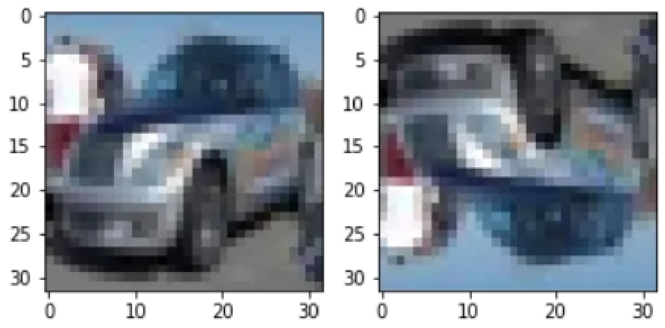
From the last train, we can see a huge improvement on the accuracy and on the loss: the test accuracy as increased and almost reaches the 80%, while the loss test doesn't stop decreasing so soon as before.

23. Does this horizontal symmetry approach seems usable on all types of images ? In what cases can it be or not be ?

The random horizontal symmetry takes an image as a input and flips it horizontally (usually in the code, with probability p). As en example:

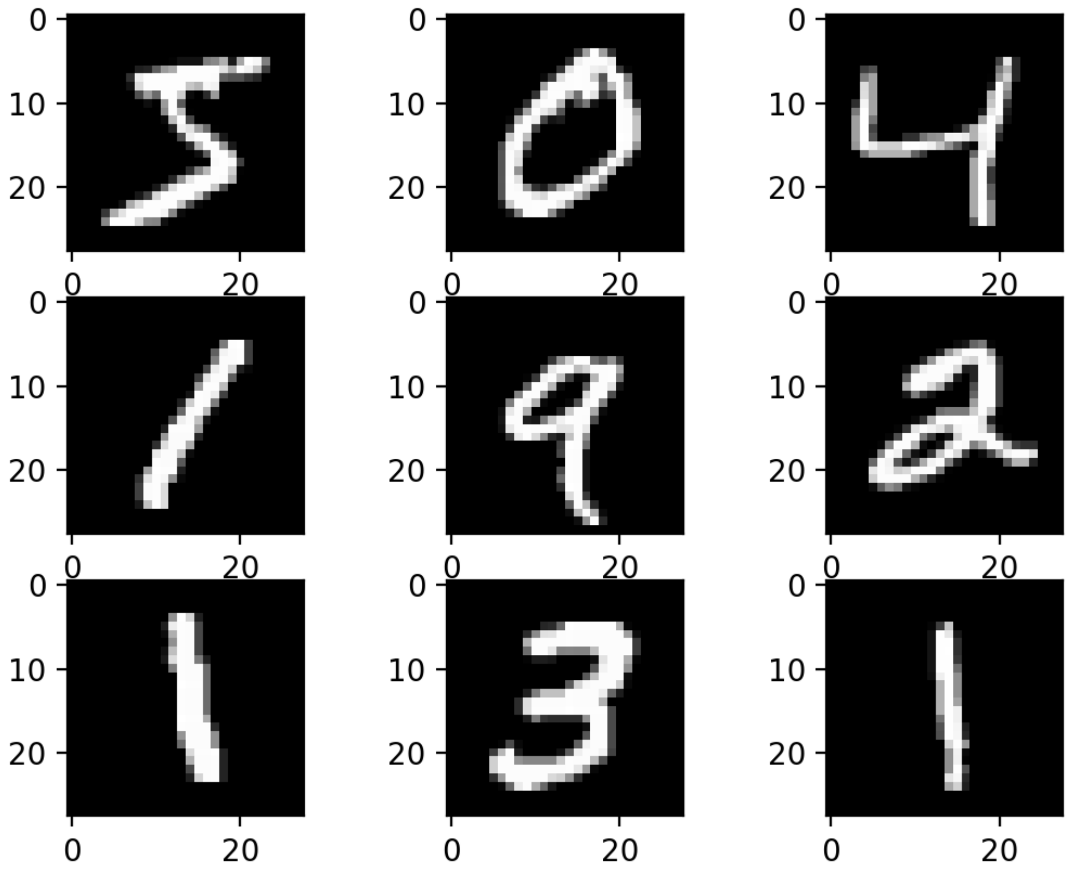


Horizontal Flip



vertical flip

Some images, like the car shown above, retain the same semantic meaning even after a horizontal flip. This consistency in semantics holds true because the other classes consist of animals or vehicles. However, a potential downside arises when dealing with images that are perfectly horizontally symmetrical: after transformation, they result in the same input image, which is indeed an unfortunate case. On the other hand, in a dataset containing letters or numbers, certain types of characters may drastically alter their semantics after flipping. For instance, consider this subset from the MNIST dataset:



The zeros and the ones (where the one is written with only a straight line) won't change their semantics, while the other numbers (5,4,9,2,3) would be unrecognizable.

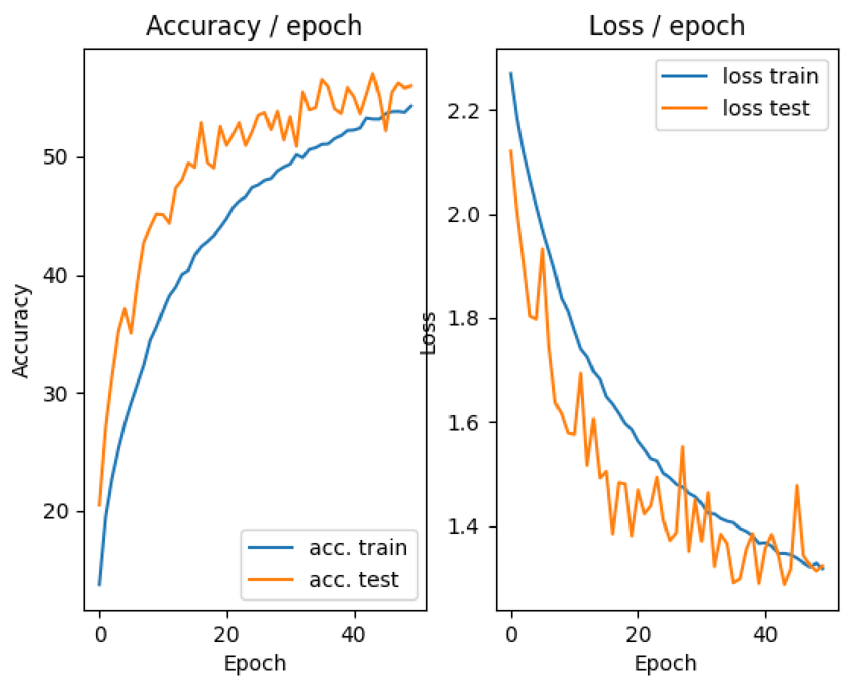
24. What limits do you see in this type of data increase by transformation of the dataset ?

As described above, it is important that the semantics of the image remain unchanged; otherwise, the network would learn parameters it shouldn't. In other words, it is essential that after a transformation, the new image remains useful for training, enabling the learning process towards the ultimate goal.

25. Bonus : Other data augmentation methods are possible. Find out which ones and test some.

Other possible transformations are adding noise to the image, performing rotations with a certain angle, making changes in brightness, saturation, using ColorJitter function for example, or partially remove a part of the image (cut out). Here we performed a test adding RandomRotation with an alpha degree of 66° and a ColorJitter, that randomly jitters the brightness in the range [0.1,0.6], jitters contrast by a factor of 1 and jitters hue by a factor of 0.4.

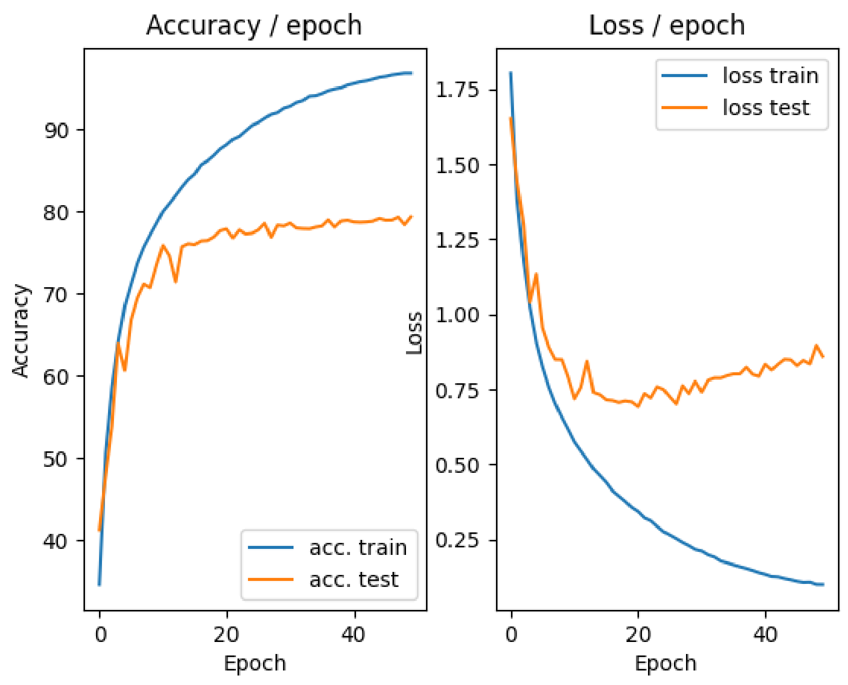
Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



3.3 Variants on the optimization algorithm

26. Describe your experimental results and compare them to previous results, including learning stability.

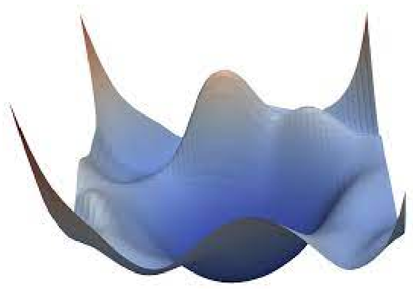
Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



The test accuracy has increased, and the test loss has decreased slightly, but still, the two curves differ significantly. Additionally, it's noticeable that the test loss has stabilized further. It's important to recall that we selected a high number of epochs to accurately represent the training results. However, by using an early stopping technique, one could opt to retain the weights that correspond to the lowest loss or the highest accuracy.

27. Why does this method improve learning ?

The optimization process of the loss function is mathematically uncertain and cannot be solved in an analytical form. This is why we rely on numerical methods or optimization schemes that consider the local gradient concerning the loss. The figure illustrates a loss landscape given our input data (where the loss function is expressed as: $L(theta_i|x_i))$. There is no guarantee that we can locate the global minimum due to the loss function not necessarily being a log-convex or a convex function.



The optimization schemes depend on several hyperparameters, including the learning rate, momentum, or Nesterov's acceleration.

As it is written here: https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler
A Learning rate schedule is a predefined framework that adjusts the learning rate between epochs or iterations as the training progresses. Two of the most common techniques for learning rate schedule are,

- Constant learning rate: as the name suggests, we initialize a learning rate and don’t change it during training;
- Learning rate decay: we select an initial learning rate, then gradually reduce it in accordance with a scheduler.

It is achieved by adding the gradient calculated in the previous step, weighted by the gamma value = 0.95, to the new gradient before the update. This allows for an increased step when the direction remains the same and a decreased step when the direction changes, thereby accelerating the learning process.

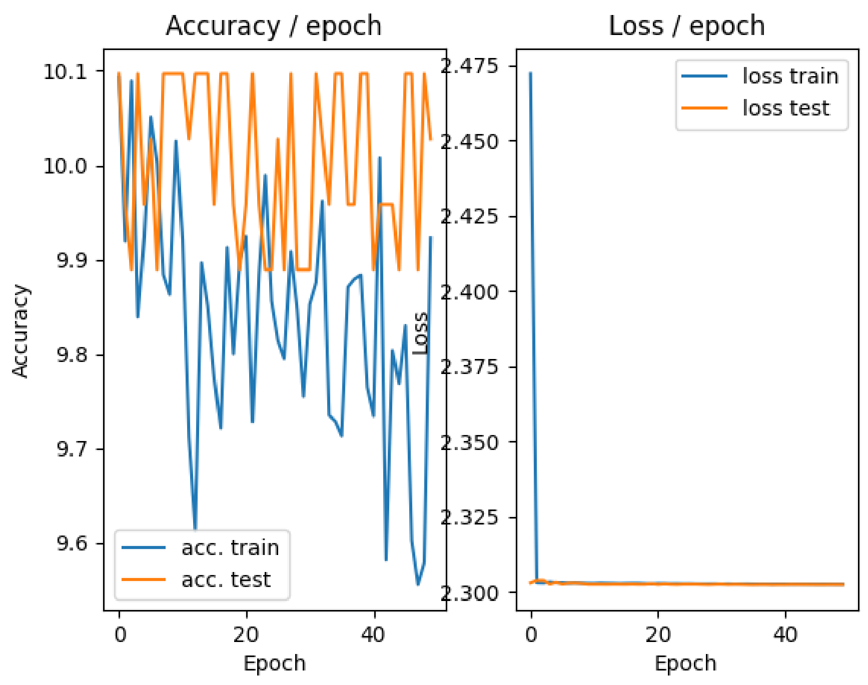
With the learning rate scheduler we can move quickly in the beginning to approach a suitable solution rapidly. Once the model has converged, reducing the learning rate allows for fine-tuned improvements.

28. Bonus : Many other variants of SGD exist and many learning rate planning strategies exist. Which ones ? Test some of them.

Other possible variants of the SDG are:

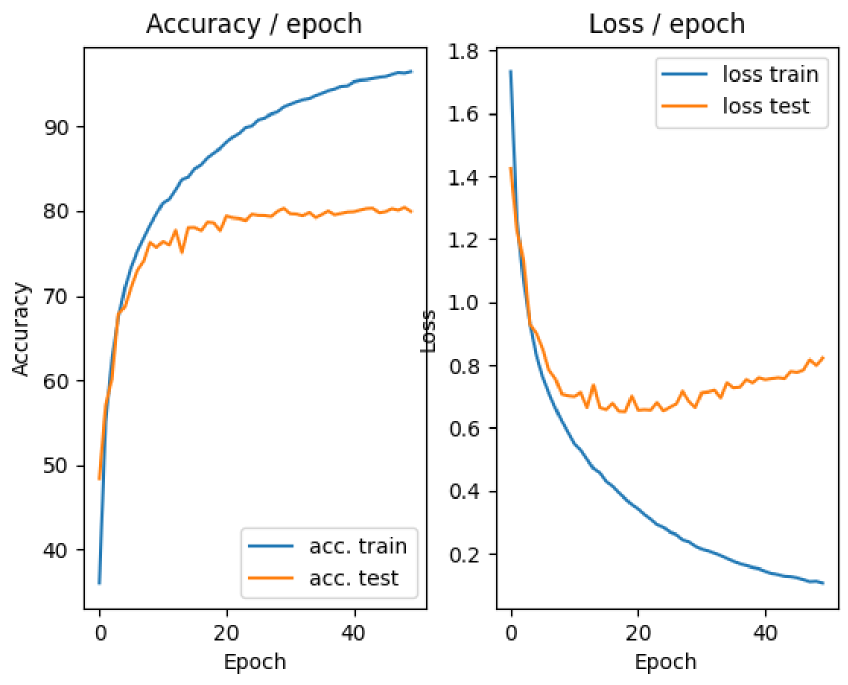
- Batch gradient descent
- Mini-Batch Gradient Descent
- Momentum Gradient Descent
- Nesterov Accelerated gradient
- AdaGrad
- RMSProp
- Adam
- ...
  Here is an experiment with Adam:

Experiment: batch_size=128, lr=0.01, epochs=50, cuda=True



In theory, it should be efficient for large datasets and noisy data. However, it requires tuning of hyperparameters and that's why here it performed so poorly.

Here is an experiment with SGD with momentum (Momentum Gradient Descent):
Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True

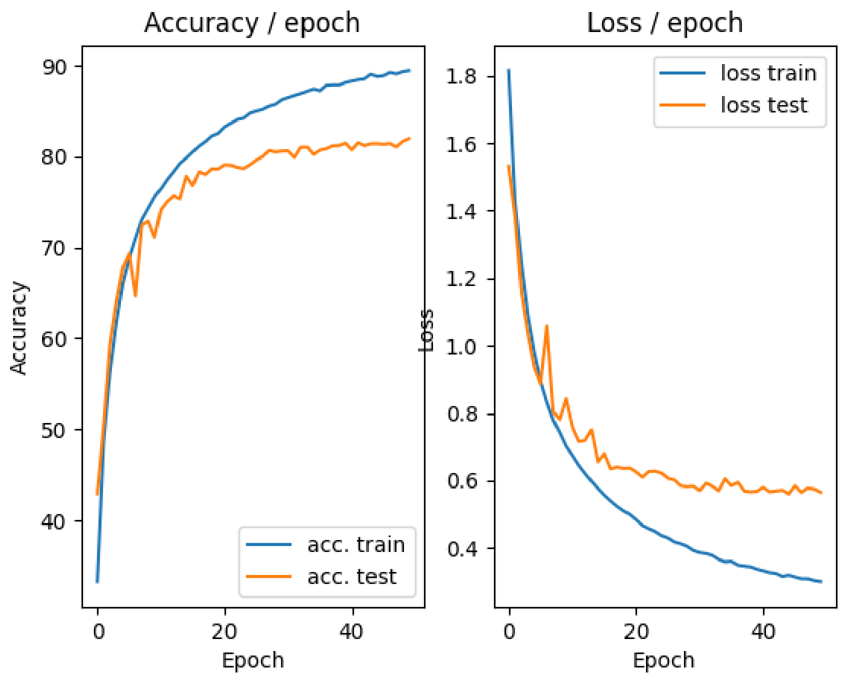


Fast convergence and it is less likely to get stuck in local minima. Compared to the SGD without momentum, the test functions have a lower fluctuation degree.

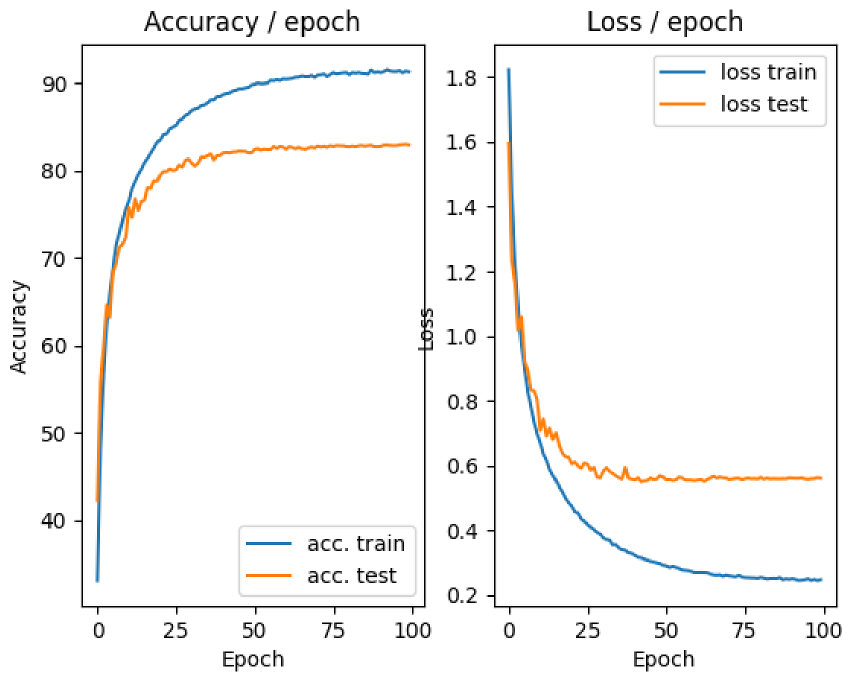
3.4 Regularization of the network by dropout

29. Describe your experimental results and compare them to previous results.

Experiment: batch_size=128, lr=0.1, epochs=50, cuda=True



After the modifications, we observed significant improvements in both the accuracy and the loss test functions. Particularly, the addition of Dropout led to a notable decrease in the overfitting phenomenon. As a network with dropout requires more epochs to train, although the epochs are computed faster for the same architecture due to fewer parameters to update, we decided to increase the number of epochs.
Experiment: batch_size=128, lr=0.1, epochs=100, cuda=True



We can see that even with a number of epochs of 100 we don't have overfitting problems.

30. What is regularization in general ?

In general, regularization is a process that penalizes the complexity of a model, and it plays a vital role in mitigating the risk of overfitting.
Regularization comes in various forms, such as L1 (Lasso) and L2 (Ridge) regularization, dropout, weight decay, and early stopping, among others. Each of these techniques applies a different form of constraint on the model's parameters, effectively curbing its complexity and promoting better generalization.

31. Research and "discuss" possible interpretations of the effect of dropout on the behavior of a network using it

Dropout is a technique that involves 'disabling' specific units in our neural network, thereby encouraging the learning of other units. During training, the gradient tends to increasingly favor certain pathways in the network, granting them more importance. By deactivating some units, the gradient is compelled to redistribute its learning across other parts of the network.

However, it's important to note that when dropout is applied, the network's expressiveness is reduced. To address this, it's often beneficial to increase the size of the neural network layers. This adjustment helps mitigate the risk of the model becoming overly reliant on specific units and enhances its ability to generalize, ultimately improving the model's overall performance and robustness.

32. What is the influence of the hyperparameter of this layer ?

The parameter 'p' corresponds to the probability of a neuron being deactivated through dropout. The choice of this probability is a crucial decision in implementing dropout effectively.

If 'p' is set too high, meaning that a substantial portion of neurons is deactivated during each training iteration, we run the risk of underfitting. Underfitting occurs when the model is too constrained and unable to capture the underlying patterns in the data, leading to poor performance.
On the other hand, if 'p' is set too low, where only a small fraction of neurons are deactivated, we may not fully realize the benefits of dropout. In such cases, the model might still be prone to overfitting, as it doesn't experience enough regularization to prevent it from learning noise present in the training data.

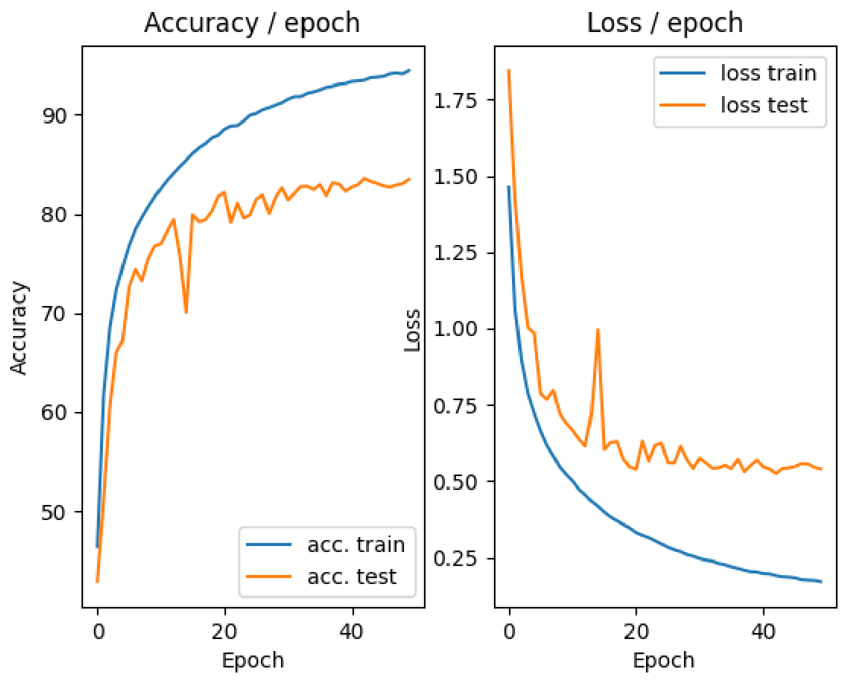
Choosing the right value for 'p' is a trade-off, and it often involves experimentation. The goal is to strike a balance between preventing overfitting and allowing the network to generalize effectively to new, unseen data.

33. What is the difference in behavior of the dropout layer between training and test ?

During the evaluation phase, dropout is disabled. Because you disable neurons *randomly*, your network will have different outputs every (sequences of) activation. This undermines consistency.

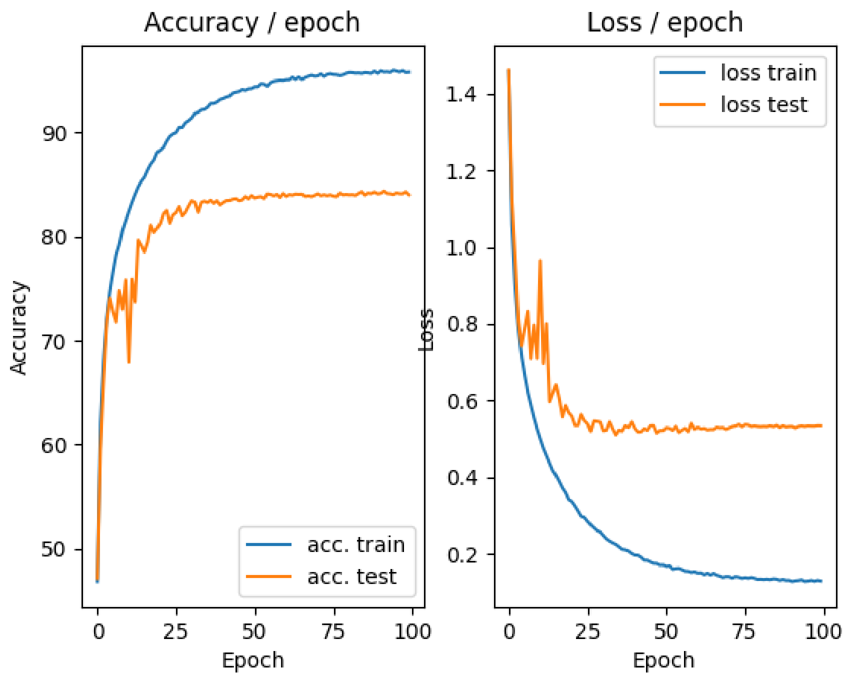
3.5 Use of batch normalization

34. Describe your experimental results and compare them to previous results.



The test accuracy improved again and also the loss has decreased again. Thanks to the dropout we don't see any overfitting effect.

We tried to use the increased number of epochs:
Experiment: batch_size=128, lr=0.1, epochs=100, cuda=True



Conclusions

Al the experiments were run on a GPU NVIDIA A100 80GB.

The initial architecture was inadequate: the accuracy was too low at 70%, and the loss exhibited some overfitting effects.

Our best model achieved an accuracy evaluation of over 80% with a loss evaluation of 0.6, without any overfitting effects. To further analyze the model, various other metrics could be used, such as precision, recall, F1 score, etc.

Another possibility is to train the model using a much larger dataset, such as ImageNet, and opt for a more complex architecture.